Challenges and Methods in Testing the Remote Agent Planner

Ben Smith
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
benjamin.smith@jpl.nasa.gov

Martin S. Feather
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
martin.s.feather@jpl.nasa.gov

Nicola Muscettola
NASA Ames Research Center
MS 269-2
Moffet Field, CA 94035
mus@ptolemy.arc.nasa.gov

Abstract

The Remote Agent Experiment (RAX) on the Deep Space 1 (DS1) mission was the first time that an artificially intelligent agent controlled a NASA spacecraft. One of the key components of the remote agent is an on-board planner. Since there was no opportunity for human intervention between plan generation and execution, extensive testing was required to ensure that the planner would not endanger the spacecraft by producing an incorrect plan, or by not producing a plan at all.

The testing process raised many challenging issues, several of which remain open. The planner and domain model are complex, with billions of possible inputs and outputs. How does one obtain adequate coverage with a reasonable number of test cases? How does one even measure coverage for a planner? How does one determine plan correctness? Other issues arise from developing a planner in the context of a larger operations-oriented project, such as limited workforce and changing domain models, interfaces and requirements. As planning systems are fielded in mission-critical applications, it becomes increasingly important to address these issues.

This paper describes the major issues that we encountered while testing the Remote Agent planner, how we addressed them, and what issues remain open.

Introduction

As planning systems are fielded in operational environments, especially mission-critical ones such as spacecraft commanding, validation of those systems becomes increasingly important. Verification and validation of mission-critical systems is an area of much research and practice, but little of that is applicable to planning systems.

Our experience in validating the Remote Agent planner for operations on board DS1 raised a number of key issues, some of which we have addressed and many of which remain open. The purpose of this paper is to share those experiences and methods with the planning community at large, and to highlight important areas for future research.

At the highest level there are two ways that a planner can fail. It can fail to generate a plan within stated time bounds[1] (*converge*), or it can generate an incorrect plan.

Plans are correct if they command the spacecraft in a manner that is consistent with accepted requirements. If the domain model entails the requirements, and the planner enforces the model, then the plans will be correct. One must also validate the requirements themselves to be sure they are complete and correct.

Ideally we would *prove* that the domain model entails the requirements: that is, prove that the model will always (never) generate plans in which particular conditions hold. This may be possible for some requirements, but is almost certainly undecidable in general.

A more practicable approach, and the one we used for RAX, is empirical testing. We first had spacecraft engineers review the English requirements for completeness and accuracy. We then generated several plans from the model and developed an automated test oracle to determine whether they satisfied the requirements as expressed in first order predicate logic. A second (trivial) oracle checked for convergence. If all of the test cases converge, and the test cases are a representative sample of the possible output plans (i.e., have good coverage), then we have high confidence that the planner will generate correct plans for all inputs.

The key issue in empirical testing is obtaining adequate coverage (confidence) within the available testing resources. This requires a combination of strong test selection methods that maximize the coverage for a given number of cases, and strong automation methods that reduce the per-test cost. Complex systems such as planners require huge numbers of test cases with corresondingly high testing costs, so this issue is particularly critical for planners.

We developed a number of test automation tools, but it still required six work-weeks to run and analyze 300 cases. This high per-test cost was largely due to human bottlenecks in analyzing results and modifying the test

[1]Since the search space is exponential there will always be inputs for which a plan exists but cannot be found within the time limit. Testing needs to show that the planner will converge for all of the most likely inputs and a high proportion of the remaining ones.

cases and automations in response to domain model changes. This paper identifies the bottlenecks and suggests some ways of eliminating them.

With only 300 cases it was impossible to test the planner as broadly as we would have liked. To keep the test suite manageable we used a "baseline testing" approach that focused the test effort on the input cases most likely to be used in operation. This strategy yields high confidence in inputs around the baseline but very low confidence in the rest of the input space. This risk is appropriate when there is a baseline input scenario that changes slowly and becomes fixed in advance of operations, as is common in space missions. Late changes to the baseline could uncover bugs not exercised by the prior baseline at a stage where there is insufficient time to fix them.

This risk could be reduced with formal coverage metrics. Such metrics can identify coverage gaps. Even if there are insufficient test resources to plug those gaps the tester can at least address the most critical gaps with a few key tests, or inform the project manager as to which inputs to avoid. Coverage metrics also enable the tester to maximize the coverage of a fixed number of tests.

To our knowledge no such metrics exist for planning systems and we did not have time to develop one of our own for testing RAX. Instead we selected cases according to an informal coverage metric. Since test adequacy could only be assessed subjectively we used more cases than were probably necessary in order to reduce the risk of coverage gaps. Formal coverage metrics for planning systems are sorely needed to provide objective risk assessments and to maximize coverage.

The rest of this paper is organized as follows. We first describe the RAX planner and domain model. We then discuss the test case selection strategy, the effectiveness of that strategy, and the opportunities for future research into coverage metrics and test selection strategies. We then discuss the test automations we employed, the demands for human involvement that limited their effectiveness, and suggest automations and process improvements that could mitigate these factors. We conclude with an evaluation of the overall effectiveness of the Remote Agent planner testing, and summarize the most important open issues for planner testing in general.

RAX Planner

The Remote Agent planner (Muscettola *et al.* 1997) is one of four components of the Remote Agent (Nayak *et al.* 1999; Bernard *et al.* 1998). The other components are the Executive (EXEC) (Pell *et al.* 1997), Mission Manager (MM), and Mode Identification and Reconfiguration (MIR) (Williams & Nayak 1996).

When the Remote Agent is given a "start" command the EXEC puts the spacecraft in a special idle state, in which it can remain indefinitely without harming the spacecraft, and requests a plan. The request consists of the desired plan start time and the current state of the spacecraft. The desired start time is the current time plus the amount of time allocated for generating a plan (as determined by a parameter, and typically between one and four hours).

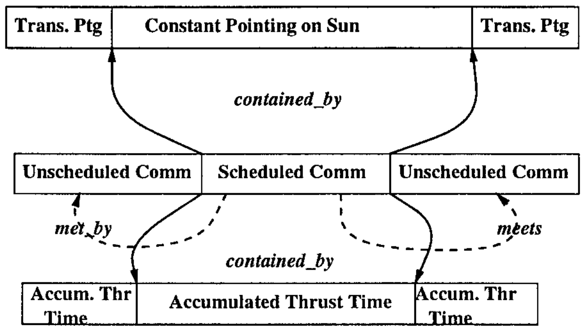

Figure 1: Plan Fragment

The Mission Manager extracts goals from the *mission profile*, which contains all the goals for the experiment and spans several plan horizons. A special *waypoint* goal marks the end of each horizon. The MM extracts goals between the required start time and the next waypoint token in the profile. These are combined with the initial state. The MM invokes the planner with this combined initial state and the absolute horizon start time, which is the requested plan start time.

The planner expands the initial state into a conflict-free plan using a heuristic chronological backtracking search. During the search the planner obtains additional inputs from two on-board software modules, the navigator (NAV) and the attitude control subsystem (ACS). These are also referred to as "plan experts." When the planner decides to decompose certain navigation goal into subgoals, it invokes a NAV function that returns the subgoals as a function of the goal parameters. The planner queries ACS for the duration and legality of turn activities as a function of the turn start time and end-points.

The fundamental execution units in the plan are tokens (activities). Tokens also track spacecraft states and resources. Tokens exist on parallel timelines, which allows activities to be executed in parallel. The plan specifies start and end time windows for each token, and temporal constraints among the tokens (before, after, contains, etc).

Nominal Execution. If the planner generates a plan the EXEC executes it. Under nominal conditions the plan is executed successfully and the EXEC requests a new plan. This plan starts at the end of the current plan, which also happens to be the start of the next waypoint in the profile.

Off-nominal Execution. If a fault occurs during execution, and the EXEC cannot recover from it, it terminates the plan and achieves an idle state. This removes the immediate threat of the fault. Depending on the failure, it may only be able to achieve a degraded idle

This selection approach required only a hundred or so cases to cover the on-board inputs that could occur during the experiment for any fixed set of ground inputs, but required an unmanageable tens of thousands of cases to cover the full space of ground inputs. Instead, we traded risk for coverage by testing just the ground inputs we thought most likely to be used for the experiment, and performing a last-minute success-oriented test on those inputs when they became available.

On-board Input Selection

The on-board inputs are the plan start time, initial state, and pseudo-random seed. Values of these inputs are partitioned into two test suites, a "replan" suite and a "back-to-back" suite. If real faults occur during the experiment, a replan could occur at any time. The initial state could be a nominal or degraded idle state. The replan suite exercises these cases.

In nominal conditions, the plan start times are at the horizon boundaries and the initial state is the final state of the previous plan. This is called "back-to-back" planning. For each back-to-back plan in the profile, the back-to-back suite exercises the possible final states of the previous plan and it's fixed plan start time (the horizon boundary). The inputs for these two suites are selected as follows.

Initial States. There are four idle states (all combinations of MICAS healthy or not, and MICAS on or off). The replan suite exercises all of them. The final states of a plan are the initial state for the following back-to-back plan. The experiment has only one such plan and it's prior plan has 14 final states. The back-to-back suite exercises all of these.

Replan start time. A replan can occur at any one-second tick between zero and the end of the experiment. The start time impacts the plan in three ways: it determines which goals the MM extracts for the planner, the proximity of those goals to the start horizon, and the length of the planning horizon. The first impact is exercised by selecting start times at the start of each goal token. This will exercise all ways of selecting goals from the profile.

The second impact primarily affect goals that require some amount of time before or after the goal for related activities. For example, the NAVIGATE token decomposes into SEP_THRUSTING activities which must be preceded by a one-hour warm-up activity. Plan start times should therefore be chosen at the boundaries of these durations.

The third impact primarily affects the SEP-related tokens. The planner first places the op-nav windows and a few other required activities. SEP is then scheduled in the remaining gaps. As the horizon shrinks, it becomes more difficult to schedule SEP activities. Identifying these horizon lengths would require a detailed analysis of the domain model, profile, and NAV goals that would have to be repeated whenever the model changed. Instead, we observed that the times selected for the other two impacts produced several different horizon lengths and accepted those as adequate.

Random Seeds. The random seeds tested were the default seed, plus two additional seeds. The default seed is always used on the first attempt, which should succeed if the planner is adequately tested. Other seeds are used only when the first attempt fails, and the EXEC makes at most five attempts. Three seeds therefore seemed reasonable.

Multiple Variation Test Cases

After selecting input parameter values, the next problem was selecting a manageable number of cases from the space of all combinations of those input values. We used orthogonal arrays (Cohen *et al.* 1996) to generate a minimal-sized test-suite in which every pair of input values appears in at least one test case, and every input value appears in about the same number of cases.

This approach detects every bug caused by a single input value or by an interaction of two input values. It will detect only some bugs caused by interactions of three or more input values. The risk of this approach is that it assumes that the majority of bugs are due to one or two input values.

The final test suite had 300 test cases. To exhaustively test all input combinations would have required over one billion cases. This reduction is possible because each case tests several pairs, and because it omits many higher-order combinations.

This set was augmented throughout the testing process with higher-order combinations that we felt were important to test. New cases were typically added to focus on faulty behaviors discovered, or hinted at, by the standard test cases.

Single Variation Test Cases.

Each multi-variation case changes several parameter values at once. When the planner failed to generate a plan, it was difficult to determine which parameters were responsible for the failure.

To address this problem, we constructed a second suite of test-cases in which each case changes only one parameter value from a baseline case that is known to generate a plan. The changed parameter value was therefore the most likely cause of the failure.

The number of single-variation test cases was equal to the sum of the parameter values, or 120 cases. In practice, these "single variation" cases caught most of the initial bugs. These cases also helped diagnose many of the failed multi-variation test cases, which often failed for the same reason as one of the single-variation cases. The remaining bugs were identified by the multi-variation cases but not by any of the single-variation cases. The high bug detection rate with a few test cases suggest that single-variation testing might be

tokens	parameters
waypoint	HZN_END, EXPT_START, EXPT_END
navigate	frequency (int), duration (int). slack (int)
Comm	none
power_estimate	amount (0-2500)
exec_activity	type, file, int, int, bool
sep_segment	vector (int), level (0-15)
max_thrust	duration (0-inf)
image_goal	target (int), exposures (0-20), exp. duration (0-15)

Table 1: Goal Tokens

state (e.g., the camera is declared broken). It then requests a new plan that achieves the remaining goals from the achieved idle state. As with other requests, the required start time is the current time plus the time allowed for planning.

Planner Inputs. The planner and mission manager can be treated as a unit for testing purposes. The inputs are the mission profile, initial state, plan start time, random seed, and plan expert outputs. The start time determines which goals the MM extracts from the profile. The other inputs are independent.

RAX DS1 Domain Model.

The domain model encodes the knowledge for commanding a subset of the DS1 mission known as "active cruise" that consists primarily of firing the ion propulsion (IPS) engine along a NAV-specified thrust arc; taking optical navigation (op-nav) images of asteroids with the MICAS camera from which NAV determines the spacecraft position; and slewing (turning) the spacecraft among image targets and thrust vectors.

Table 1 lists the goal tokens. The sep_segment and max_thrust_time tokens specify the thrust arc. The segments specify the direction and level of any thrust contained by that token and max_thrust_time specifies the desired thrust time. The navigate token determines the duration and periodicity ($\pm$ a "slack" value) of the op-nav windows, and the take_image_goal tokens specify the image targets. The comm(unication) tokens specify periods when the spacecraft must have the low-gain antenna Earth-pointed. The exec_activity goals specify simulated faults that EXEC should inject into RAX. These were added to demonstrate RAX's fault recovery capabilities since real faults were unlikely to occur during the experiment. The power_estimate is the power level to use for planning, and waypoints delineate horizon boundaries.

Table 2 shows the initial states. NO_ACTIVITY token specifies the last exec_activity token executed to avoid planning an executed goal during a replan. The attitude token specifies the initial attitude. The MICAS_HEALTH token specifies whether the MICAS power switch is broken or healthy, and the MICAS_MODE token determines whether it is on or off. If MICAS is stuck the plan cannot change the switch state. If it is stuck-off the plan cannot take images. The other initial state tokens are fixed.

tokens	parameters	N
NO_ACTIVITY	counter $\in \{0, 1, 2\}$	3
ACCUM_THRUST_TIME	all fixed	1
IDLE_SEGMENT	none	1
TIMER_IDLE	none	1
SEP_STANDBY	none	1
CONSTANT_POINTING	target $\in$ { Earth, image, thrust vector }	3
MICAS_IDLE	none	1
MICAS_READY or MICAS_OFF	none none	2
MICAS_HEALTH	healthy $\in$ true, false	2
INACTIVE	none	1
NAV_IDLE	none	1
PLANNER_IDLE	none	1
Combinations		36

Table 2: Initial State Tokens

Test Selection Strategy

The key test selection issue is achieving adequate coverage with a manageable number of cases. Test selection should ideally be guided by a coverage metric in order to ensure test adequacy. Coverage metrics generally identify equivalence classes of inputs that result in qualitatively similar behavior with respect to the requirement being verified. A set of tests has full coverage with respect to the metric if it exercises the test artifact on one input from each class.

The verification and validation literature is full of coverage metrics for mission-critical systems (e.g., code coverage), but to our knowledge there are no coverage metrics specifically suited to planning systems. The most relevant metrics are those for verifying expert system rule bases. The idea is to backward chain through the rule base to identify inputs that would result in qualitatively different diagnoses (e.g., (O'Keefe & O'Leary 1993)). Planners have more complex search engines with correspondingly complex mappings, and a much richer input/output space. It is unclear how to invert that mapping in a way that produces a reasonable number of cases.

For the RAX planner we used a very straightforward, informal version of this strategy. For each input parameter we selected values at extrema (e.g., low, middle, high) or values that we thought would produce qualitatively different plans or paths through the search space. Although only a few values of each parameter are selected, it was still impractical to test all combinations of those values (there are millions). Two methods termed *multi-variation* and *single-variation* selected a subset of these combinations for testing.

a good approach for applications with a high risk tolerance and limited testing resources.

Ground Input Selection

The ground inputs consist of the mission profile, planner parameters (such as the planning duration), and parameter settings for ACS and NAV. The most important of these inputs is the one used for the experiment itself. Planner testing had to ensure that the planner would meet its requirements for all on-board inputs it could receive with this single ground input in effect.

Without a formal metric to show how the profiles and other ground inputs would impact the output plans, it would have required thousands of cases to feel confident in the coverage. Since this was unmanageable, we focused on the ground inputs most likely to be used in flight. The test cases consisted of the current set of expected inputs (the baseline) and the most likely future changes. As the experiment approached, the baseline become better defined and the test cases were updated accordingly. The inputs were finalized one month before the experiment at which point we tested them against all possible replan times. Since there would be little time to fix any bugs detected at this point the prior testing had to provide high confidence that these last-minute tests would pass. We termed this approach "baseline testing."

The baseline inputs were as follows. The chosen ACS slew durations, NAV image parameters, and SEP thrust levels were low, medium, and high values within the range of expected values. The remaining plan expert input, the NAV thrust arc, is specified as a series of sep_segment tokens. We exercised zero to three segments with several relative placements, and appealed to induction to cover four or more segments. We also changed the goals in the profile. We changed the duration and placement of the op-nav windows, the power level, and the absolute experiment start time. The exec activities were not varied since they were under our control. We intended to test comm goal variations, but dropped this after early indications that we would have full antenna coverage during the experiment and could therefore eliminate the goals or set them as we wished (we did the later).

Risks of Baseline Testing. Baseline testing assumes that the final baseline will be very close to the last tested baseline, and that any last-minute changes will have been covered by the most recently tested variations. This approach is vulnerable to radical changes made close to execution. Radically different inputs will not have been adequately tested, and there will be less time to address any bugs that occur at this stage. If changes occur very close to execution, and the new baseline uncovers new bugs, there will not be time to fix them. The execution would have to be delayed, or it would have to proceed with the bugs present. Project mangers must be made aware of these risks and be prepared to address these contingencies if they occur.

Version	days	tokens	parameters		relations	
			chg	new	chg	new
009	6	27		34		13
011	6		1		6	
015	6		0		1	
019	6		0		10	-1
026	6	+8	1	+9	8	+9
029	6			+5	0	
FLT 03	6		1		0	
FLT 05	2	-12,+3	10	0	15	-7
FLT 07	2		5		7	+2

Table 3: Profile Evolution

The RAX baseline was relatively stable for over a year. Table 3 summarizes the evolution of the RAX baseline profile. Over this period the only change to the goal tokens were an addition of four comm goals (and four "no_comm" goals between them) in version 026; an additional waypoint token parameter and three navigate token parameters in 026; and an additional exec_activity parameter in 029. The relative placement of the goals remained stable, although the absolute placement changed frequently.

Beginning in January of 1999 RAX experienced two late baseline changes. The first change occurred in January when we integrated the planner with the real plan experts, which had just become available. Until then we had been testing with simulated experts. The real experts turned out to have different ranges than the simulators. For example we assumed turn durations were at most 20 minutes, but in fact some turns could take over an hour. The requirements were either incorrectly captured to begin with, or had changed unbeknownst to the RAX team. Since the experts were available later than originally expected, there was now no time in the schedule to rerun the full test suite with the new baseline. We had to be content with a handful of tests that exercised the new ranges.

The second change occurred in March, two months before the experiment. For operational reasons RAX was no longer allowed to turn off the MICAS camera and had to reduce IPS thrusting from five days to under twelve hours. The baseline profile changed from six days to two, deleted an exec activity goal and four comm goals, changed several navigate goal parameters, and changed the absolute placement of the goals. The goal definitions, relative placement, and overall structure remained the same. Less radical changes occurred over the next month as the baseline stabilized (navigate token parameters and temporal relations). The original baseline tests apparently covered this space well enough that only two minor bugs were detected in the final testing of the new baseline. This is probably because the basic structure of the two baselines were similar, even though several major changes were made.

Test Effectiveness

This selected tests were ultimately successful in validating the planner in that they provided sufficient confidence for the DS1 project to approve the Remote Agent Experiment for execution on DS1, and the on-board planner exhibited no faults during the experiment. A total of 211 bugs were reported.

Since there were no formal coverage metrics, it was difficult to assess test adequacy. Some 22 problems, or a little over 9% of the total problem reports, were discovered during integration and development but would not have been caught by the official test suite. These problems provide useful insight into the coverage gaps.

1. Planning problems became more challenging when we transitioned from the 6 day scenario to the 2 day scenario. The temporal compression led to the disappearance of slack time between activities. In the 6 day scenario PS could exploit this slack to achieve subgoals without backtracking. In the 2 day scenario backtracking became necessary, revealing additional brittleness in the PS chronological backtracking search.
2. In at least one case the test selection missed key boundary values that would have been apparent with a more detailed model analysis. This problem depended upon the specific values of three continuous parameters: the time to start up the IPS engine, the time to the next optical navigation window, and the duration of the turn from the IPS attitude to the first asteroid. An equation relating these parameters can crisply identify the boundary values that should be exercised.

These coverage gaps could have been detected if better coverage metrics had been available to guide test selection. The test selection was based on an informal, high level analysis of the model, and doubtless missed many such subtle interactions.

Formal Coverage Metrics Needed

Formal coverage metrics are sorely needed for planner validation. We required thousands of cases to cover the ground inputs, especially the goals, because there was no formal analysis to indicate which input combinations needed to be tested and which could be ignored. A good metric would have allowed us to eliminate unnecessary combinations confidently and determine where additional cases were needed.

Formal metrics provide can identify coverage gaps and can inform cost-risk assessments. If one knows how many cases are needed for a given level of coverage (risk), one can make an informed decision on how to balance the number of cases (cost) against coverage (risk).

Formal coverage metrics, such as code coverage, have been developed for critical systems but to our knowledge no metrics have been developed for measuring coverage of a planner domain model. This is clearly an area for future research. A few possibilities are discussed below.

Constraint coverage. One possible coverage metric is the number of compatibilities covered. This is analogous to a code coverage metric. For a given plan, it determines which compatibilities (constraints) it uses, and how those compatibilities were instantiated. A good test suite should exercise each instantiation of each compatibility at least once.

Goal-Interaction coverage. This coverage metric is targeted at exercising combinations of strongly interacting goals. Since testing all combinations is intractable, the idea is to analyze the domain model to determine how the goals interact, and only test goal combinations that yield qualitatively different conflicts. For example, if goals A and B used power, we would test cases where power is oversubscribed by several A goals, by several B goals, and by a combination of both goals.. The coverage could be adjusted to balance risk against number of cases. One could limit the coverage to interactions above a given strength threshold.

This metric would extend on prior work on detecting goal interactions in planners to improve up the planning search, such as STATIC (Etzioni 1993), Alpine (Knoblock 1994) and Universal Plans (Schoppers 1987). STATIC generates a problem solving graph from the constraints and identifies search control rules for avoiding goal interactions. Alpine identifies interactions to find non-interacting sub-problems, and universal plans (Schoppers 87) derive reactive control rules from pair-wise goal interactions. These methods are designed for STRIPS-like planning systems and would have to be extended to deal with metric time and aggregate resources, both of which are crucial for spacecraft applications. One of the authors (Smith) is currently pursuing research in this area.

Slack metric. Another approach being pursued by one of us (Muscettola) is to select plan start times by analyzing the slack in the baseline plans. This approach was used to manually select plan start times once the final baseline was frozen just prior to the experiment.

Using our knowledge of the PS model, we manually identified boundary times at which the topology of the plans would change. We identified 25 such boundary times and generated a total of 88 test cases corresponding to plans starting at, near, or between boundary times. This led to the discovery of two new bugs. Furthermore, analysis of the test results showed that PS would fail to find a plan at only about 0.5% of all possible start times. Although the probability of this failure was extremely low, contingency procedures were developed to ensure that the experiment could be successfully continued even if this PS failure actually occurred.

Test Automation

Automation played a key role in testing the Remote Agent planner. It was used for generating tests, running tests, and checking test results for convergence and plan correctness. Even so, the demand for human involvement was high enough to limit the number of test cases to three hundred per six week test period, or an average of ten cases per work-day.

There were two main bottlenecks where human involvement was required: analysis, and changes to the test suite and automation infrastructure caused by changes to the model and baseline. This section discusses the automations that we found effective, the human bottlenecks, and opportunities for further automation.

Testing Tasks

The Remote Agent software, including the planner, was released for testing every six to eight weeks. The planner was exercised on the full set of test cases. A typical test cycle consisted of the following activities.

The tester updates the set of test cases as required by any changes to the baseline, goal tokens, or initial state tokens. A test harness invokes the planner on each test case and collects the output. If the test cases exercise new input parameters, or planner interfaces have changed, the harness must be upgraded and debugged first. The tester makes sure that the plans ran properly, and re-runs any that failed for irrelevant reasons (e.g., the ACS simulator did not start).

The test results are analyzed by two oracles. The first checks for convergence, and the second for plan correctness. The oracles say that a requirement failed, but not why it failed. The tester reviews the output to determine the proximate cause and files a bug report.

Finally, the analyst confirms purported bug fixes from the previous release as reported in the bug-tracking database. Each bug has one or more supporting cases. The analyst determines whether those cases passed, or whether the bug is still open. In some instances, the tester may have to devise additional tests to confirm the bug fix.

Test Automation Tools.

We employed several test automation tools for validating the Remote Agent planner, which are summarized below.

- **Test case generator.** This tool generated a manageable number of test cases from the cross product of all selected parameter values. It first selected the input parameters that would comprise each test case, according to the single-variation and multi-variation (orthogonal arrays) methods described earlier. It then converted the parameter values to input files: initial state, planner parameters (random seed), and a parameter file that governs the ACS and NAV simulators. The mission profiles were too difficult to generate automatically and were constructed by hand.
- **Test Harness.** The harness invokes the planner with the inputs for a given test case and collects the output, which consists of the plan file (if any), time spent planning, search trace, the initial state generated by the mission manager, and the simulator and harness output.

Task	Effort
Update/debug cases, tools	3.0
Run cases and analyzers	0.1
Review analyzer output	1.5
File bug reports	0.5
Close bugs	0.5
Total	5.6

Table 4: Test Effort in Work Weeks by Task

- **Plan Correctness Oracle.** The oracle reads a plan into an assertions database and then verifies that the assertions satisfy requirements expressed in first order predicate logic (FOPL). This tool (Feather 1998; Feather & Smith. 1999) was implemented in AP5, a language that supports these kinds of FOPL operations.

  The oracle also verified that the plan engine enforced the plan model by automatically converting the plan model into equivalent FOPL statements and checking the plan against them. Compatibilities are of the form "if token A exists in the plan, then there also exists a token B such that the temporal relation R holds between A and B." This maps onto an equivalent FOPL requirement: $A \rightarrow B \wedge R(A, B)$.

Human Bottlenecks

The biggest demand for human involvement was from changes to the model and baseline that required modifying the test cases and tools. The next largest demand was for reviewing the output from the automated requirement checkers as a prelude to filing bug reports. The test effort by task is shown in Table 4.

Impact of Model and Baseline Changes. About half of the test effort in each cycle were the result of changes to the model, baseline, and planner interfaces. These changes required modifications to the composition of the test suite, and modifications to the test harness. Table and shows how the model evolved over several releases, and Table 3 in the Test Selection section shows how the baseline evolved.

When new input parameters are added, the harness must be updated to process them appropriately. These changes are fairly straightforward, and can typically be completed and debugged in about a day.

A more time-consuming impact comes from tracking changes to planner interfaces. Most of the test parameters are converted into planner input files which the harness feeds to the planner. The initial state, simulator parameters file, and the planner parameters file are created automatically; the mission profiles are created manually. When the syntax or semantics of these files changed, or the input parameters changed, the input

version	date	compats	relations	disjuncts
009	4/23/98	43	121	19
011	7/09/98	40	120	19
015	8/23/98	39	116	19
019	9/07/98	37	111	19
026	10/02/98	39	207	19
027	11/05.98	41	237	23
029	12/16/98	46	261	26
FLT 003	2/07/99	46	257	26
FLT 005	3/19/99	46	261	26
FLT 007	4/08/99	46	259	27

Table 5: Model Evolution

files had to be regenerated.

Although updating the input file generator and regenerating the files only took a day or so, debugging the resulting test cases often took several days. If a test case failed, it could have been because of a real bug, or because of an error in the input file. Some of these were obvious, and detected in a dry run with a few test cases. Others were more subtle and occurred only under certain conditions. These were not detected until the analysis phase, at which point the cases had to be re-run and re-analyzed.

This problem was exacerbated by undocumented interface changes. Many of the planner interfaces were internal to the Remote Agent. Changes were sometimes omitted from the release notes, and would not become apparent until the test cases were analyzed.

This experience indicates that it is preferable to maintain stable interfaces throughout testing if at all possible. If interfaces must change, they should do so infrequently and the decision to change them should factor in the test impact.

However, two key interfaces are particularly resistant to this policy: the initial state and the mission profile. These files are comprised of tokens, and if these token definitions change in the domain model, these input files must also change. Table shows how the token parameters changed over time. The number of token types remained constant.

We observed that token parameters were most often added in order to propagate values for use by compatibilities or heuristics and did not record values specified externally. We created the notion of a *private* parameter to avoid changes. Private parameters do not appear in the initial state or profile, but are added automatically by the MM. Their values are set automatically by propagation from other parameters. This reduced the number of impactful parameter changes from 30 to 10.

We addressed the initial state problem by negotiating an interface to the initial-state generating function in the EXEC code. The test harness constructed an initial state by sending appropriate inputs to those functions, which would then create the initial state that was identical to the one used without the harness.

Token	Type	Public	Private	Version
navigate	goal	+3		026
op nav window	goal		+8	026
no op nav	goal		+4	026
waypoint	goal		+1	026
no op nav	goal		+1	027
op nav window	goal		+1	029
exec goal	goal	+1		029
sep timer idle	init		−1	019
SEP standby	init		+1	019
no activity	init		+1	029
micas health	init		+1	029
MICAS ready	init		−1	029
RCS turn	internal	+5		019
sep turn	internal	+1		026
exec activity	internal		+1	029

Table 6: Token Parameter Changes

With baseline testing approach, the baseline profile and the most likely variants are *expected* to change over time, and in practice changed with each test release. The baseline profile is known to work, since that is the acceptance criteria for delivering the software for testing. The variant profiles are created by hand from the baseline profile. It is difficult to generate valid profiles (ones with satisfiable goals), since it requires some knowledge of how the goals interact with the domain model. The debugging effort for the mission profile variants could have been greatly reduced by automatically checking the profile validity. One could imagine automating these checks by using an abstraction of the domain model to prove that a set of goals are unsatisfiable.

The debugging effort for other inputs could have been reduced in a similar fashion. The syntax and semantics of each input file could be formally specified and automatically verified against that specification. This would have detected interface changes (the input files would be invalid) and eliminated most of the debugging effort by detecting input file inconsistencies early and automatically.

Changing Requirements. Whenever the requirements changed, the automated requirement checkers had to be updated accordingly. The requirements documents often lagged development, so requirement changes were sometimes not detected until the bug reporting phase. The documents and automated checkers then had to be updated. In some cases, several test cases had to be re-analyzed against the changed requirement.

It is almost impossible to get all the requirements right in the first draft of the requirements document. A certain amount of inefficiency and missed requirements should be expected for the first few test cycles as the document is refined. It may be possible to converge faster by having the developers to run the automated plan checker on the baseline scenario before releasing the software for testing. This will highlight the obvious discrepancies between the formal requirements and the developer's expectations.

Running Tests Unnecessarily. When the domain model changes it may not be necessary to run all the tests again. Some test results from the previous version of the model may still be valid. We had no way to identify these "unimpacted" tests confidently, and therefore had to run all of the tests in every test cycle. If a reliable method were available, it could significantly reduce the number of tests that had to be run in each test cycle with no increase in risk. One could imagine analyzing the domain model to determine whether the test case would exercise the changes. For example, a test case in which the MICAS camera is stuck off would not exercise a change to the duration of the MICAS_ON token.

This capability would also allow one to assess the cost of testing proposed model changes. This is an important factor in deciding how (or even whether) to fix a bug near delivery, and in assessing which fixes or changes to include in a release.

Analysis Costs. The oracles identify cases that do not meet the requirements, but do not explain why they failed. For each failed test case, the analyst determines the proximate cause of the failure and groups cases that have similar causes. This diagnosis provides a strong hint for finding the underlying bug, and is critical for tracking progress. If the analyst simply stated that the planner failed to generate a plan on the following fifty test cases, that could mean there are fifty underlying bugs or just one. The first requires far more work than the second. The initial diagnoses provide a much better estimate of the number of outstanding bugs.

These analyses took eight to ten work-days for a typical test cycle and were largely unautomated. To determine why a plan failed to converge required the tester to examining the plan search trace and consider how the test case differs from cases that do not exhibit the behavior. Plan correctness failures require similar review, although it is somewhat simpler (2-3 days) since the failure is isolated and the oracle identifies the offending plan elements.

Automated diagnosis could reduce these efforts, especially for determining why the planner failed to generate a plan. There has been some work in this area that could be applied or extended. Howe (Howe & Cohen 1995) performed statistical analyses of the planner trace to determine which combinations of state and repair operator were abnormally likely to cause failures. Chien (Chien 1998) allowed the planner to generate a plan, when it was otherwise unable to, by ignoring problematic constraints. Analysts were able to diagnose the underlying problem more quickly in the context of the resulting plan.

Conclusions

The main requirements for the Remote Agent planner were to generate a plan within the time limit, and that the plan be correct. The validation approach for the Remote Agent planner was to invoke the planner on several test cases, and automatically check the results for convergence and plan correctness. Correctness was measured against a set of requirements developed by the planning team and validated by system and subsystem engineers. The cases were selected to exercise key boundary points and extrema in the mission profile (goals) and domain model. These values were identified informally, based on the tester's knowledge of the domain model.

Our informal coverage metrics required only a few hundred cases to obtain good coverage of the inputs that would occur on-board for any given set of goals, but required thousands of cases to cover the entire goal space. This was unmanageable given the available workforce, even with automated plan checkers and test runners.

Analysis costs were high because of the need to provide initial diagnoses for cases where the planner failed to generate a plan, and the need to review the plan checker's output. Changes to the planner interfaces, including changes to the model, also created an overhead for updating and debugging the test harness. Changing requirements imposed a similar overhead on the automated requirement checkers. We suggested a number of ways to mitigate these factors.

Since we could not manage the number of cases needed to adequately cover the entire goal space, we focused our efforts. The planner only had to work on one set of goals—those used in flight—but those goals would not be finalized until a few weeks before the experiment. We therefore focused our cases on the current baseline goals and few likely variations to it. This exercised the planner on inputs close enough to the final baseline that when the final baseline was tested just before the experiment only a few minor bugs were found.

Chasing an evolving baseline imposed an overhead of updating and debugging the test suite and test harness when the baseline changed. It was also vulnerable to late, radical changes to the baseline despite Remote Agent's success with such changes. This approach should only be used with full knowledge of the inherent risks and limitations.

A better approach would be to exercise the goal space more completely. This would almost certainly require far more test cases than we were able to manage for the Remote Agent planner. There are a number of open research opportunities in this area. Formal coverage metrics are sorely needed for planners. Such metrics could guide the test selection and inform decisions on balancing risk (coverage) against cost (number of cases). Clever metrics may also be able to reduce the number of cases needed for a given level of coverage as compared to the straightforward metrics used for the Remote Agent planner.

The number of manageable cases could also be increased by reducing the demand for human involvement. Automated diagnosis methods would eliminate one bottleneck, especially methods for determining why no plan was generated. Methods for identifying illegal inputs, especially illegal goals and initial states, would eliminate some of the test-case debugging effort, as would process improvements for limiting interface changes.

The Remote Agent was a real-world, mission-critical planning application. Our experience in validating the Remote Agent planner raised a number of key issues. We addressed several of these, but many issues remain open. As planning systems are increasingly fielded in critical applications the importance of resolving these issues grows as well. Hopefully the Remote Agent experience will spark new research in this important area.

Acknowledgments

This paper describes work performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract from the National Aeronautics and Space Administration, and by the NASA Ames Research Center. This work would not have been possible without the efforts of the rest of the Remote Agent Experiment team and our assistant testers Todd Turco and Anita Govindjee.

References

Bernard, D.; Dorais, G.; Fry, C.; Gamble, E.; Kanefsky, B.; Kurien, J.; Millar, W.; Muscettola, N.; Nayak, P.; Pell, B.; Rajan, K.; Rouquette, N.; Smith, B.; and Williams, B. 1998. Design of the remote agent experiment for spacecraft autonomy. In *Proceedings of the 1998 IEEE Aerospace Conference.*

Chien, S. 1998. Static and completion analysis for knowledge acquisition, validation and maintenance of planning knowledge bases. *International Journal of Human-Computer Studies* 48:499–519.

Cohen, D.; Dalal, S.; Parelius, J.; and Patton, G. 1996. The combinatorial design approach to automatic test generation. In *IEEE Software*, 83–88.

Etzioni, O. 1993. Acquiring search control knowledge via static analysis. *Artificial Intelligence* 62:255–302.

Feather, M., and Smith., B. 1999. Automatic generation of test oracles: From pilot studies to applications. In *Proceedings of the Fourteenth International Conference on Automated Software Engineering (ASE-99)*, 63–72. Cocoa Beach, FL: IEEE Computer Society. Best Paper.

Feather, M. 1998. Rapid application of lightweight formal methods for consistency analysis. *IEEE Transactions on Software Engineering* 24(11):949–959.

Howe, A. E., and Cohen, P. R. 1995. Understanding planner behavior. *Artificial Intelligence* 76(2):125–166.

Knoblock, C. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* 68(2).

Muscettola, N.; Smith, B.; Chien, C.; Fry, C.; Rajan, K.; Mohan, S.; Rabideau, G.; and Yan, D. 1997. Onboard planning for the new millennium deep space one spacecraft. In *Proceedings of the 1997 IEEE Aerospace Conference*, volume 1, 303–318.

Nayak, P.; Bernard, D.; Dorais, G.; Gamble, E.; Kanefsky, B.; Kurien, J.; Millar, W.; Muscettola, N.; Rajan, K.; Rouquette, N.; Smith, B.; Taylor, W.; and Tung, Y. 1999. Validating the ds1 remote agent. In *International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS-99).*

O'Keefe, R., and O'Leary, D. 1993. Expert system verification and validation: a survey and tutorial. *AI Review* 7:3–42.

Pell, B.; Gat, E.; Keesing, R.; Muscettola, N.; and Smith, B. 1997. Robust periodic planning and execution for autonomous spacecraft. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97).*

Schoppers, M. 1987. Universal plans for reactive robots in unpredictable environments. In *IJCAI 87.*

Williams, B., and Nayak, P. 1996. A model-based approach to reactive self-configuring systems. In *Proceedings of the thirteenth national conference on artificial intelligence (AAAI-96)*, 971–978.